Ciencias computacionales

Propedeutico: Programación

INAOE

Contents

1 Clases y objetos

Material multimedia recomendado:

- Clases: `https://www.youtube.com/watch?v=lBhSrZd7oW8`
- Herencia:
  - `https://www.youtube.com/watch?v=xmoKYupEUO8`
  - `https://www.youtube.com/watch?v=s5DFILGL7Jo`
- Introducción al polimorfismo `https://www.youtube.com/watch?v=6lIGfzZ4oqo`

En esta sección se presenta un resumen de la programación orientada a objetos. Se presentarán algunos conceptos básicos como son: clases, objetos, atributos, métodos, herencia y polimorfismo.

1.1 Definición

Un programa orientado a objetos sólo se compone de objetos, los cuales pueden contar con un estado interno y mandar mensajes a otros objetos a través de su interfaz pública. Un objeto pertenece a una clase y cuando es creado se dice que esta instanciando esta clase. A continuación de define lo que es una clase y sus miembros:

- Una *clase* es un tipo definido por el usuario que describe los atributos y métodos de los objetos que se crearán a partir de la misma.
- Los *atributos* definen el estado interno de un objeto.
- Los *métodos* son las operaciones que definen su comportamiento.

Los constructores que permiten inicializar un objetos, y los destructores que permiten destruirlo forman parte de los métodos.

En la figura 1 se muestra un representación gráfica de un objeto.

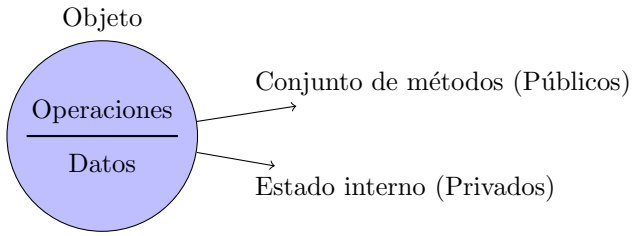

Figure 1: Objeto

También se le conoce como miembros de la clase al conjunto de métodos y atributos.

En el siguiente código se muestra la definición de una clase. La cuál consta de la palabra reservada **class**, el *nombre de la clase*, y *el cuerpo de la clase*.

```
1        class nombre_clase{
2                //cuerpo de la clase
3        };
```

El cuerpo de una clase generalmente consta de atributos y métodos, los cuales constan de modificadores de acceso.

1.2 Miembros (funciones y atributos)

1.2.1 Atributos

La estructura interna de los objetos de una clase esta definida por sus *atributos* (en C++ también conocidos como *dato miembro*).

Un atributo se declara de igual forma que se declara cualquier otra variable, pero esto debe hacerse dentro del cuerpo de la clase. es recomendable declarar primero los atributos y despues los métodos. Por ejemplo:

```
1 class Alumno{
2         private:
3                 string nombre;
4                 int edad;
5 };
```

La clase Alumno consta de dos atributos *nombre* y *edad*. El modificador de acceso *private* se explicara más adelante.

Los atributos de una clase deben ser unicos, no se permite declarar dos atributos con el mismo nombre dentro de una clase. Sin embargo, se puede utilizar el mismo nombre de atributo dentro de otra clase. Cada clase define un espacio de nombre. De esta forma, la clase Alumno tiene un atributo edad, y alguna otra clase tambien puede tener un atributo llamado edad.

A diferencia de otros lenguajes, en C++ no es posible asignar un valor inicial a un atributo de una clase.

Por ejemplo:

```
1 class Alumno{
2         private:
3                 string nombre;
4                 int edad = 15;  //error: inicialización no permitida
5 };
```

Normalmento la inicalización de un objeto se realiza por medio de su constructor.

Un objeto de una clase existente puede ser declarado como atributo de alguna otra clase. Esto se muestra en el siguiente ejemplo:

```
1 class Alumno{
2         private:
3                 Direccion dir; // Clase previamente declarada
4                 string nombre;
5                 int edad;
6 };
```

El atributo dir es un objeto de la clase Dirección, la cual contiene sus propoios atributos y métodos.

Una clase no puede contener un objeto de su misma clase, a no ser que sea declarado como static o puntero. Por ejemplo:

```
1 class Alumno{
2         private:
3                 Alumno peor_enemigo; // Error
4                 Alumno *mejor_amigo; // Correcto
5                 string nombre;
6                 int edad;
7 };
```

1.2.2 Métodos

Los métodos son los que generalmente dan acceso a la estructura interna (datos, atributos) de los objetos. También definen las operaciones que un objeto puede realizar, y desde el punto de vista de la Programación

Orientada a Objetos la **interfaz** que pueden usar otros objetos para mandarle un mensaje. En C++ algunos autores también los llaman **funciones miembro** de la clase.

Los métodos se declaran dentro de su clase. No es posible anidar métodos, esto es, no puede ir un método dentro de otro. Para declarar un método se deben seguir todas las recomendaciones del capítulo de funciones.

1.2.3 Control de acceso

La Programación Orientada a Objetos incluye la idea de ocultación de datos, que básicamente consiste en que no se pueden acceder directamente a los atributos de un objeto, si no que hay que hacerlo a través de sus métodos públicos (también conocidos como métodos de acceso). De esta forma solo se tiene acceso a algunos métodos públicos de los objetos para poder acceder a sus atributos privados. Con esto, se consiguen dos objetivos importantes:

- Que el usuario no genere código basado en la estructura interna de una clase, ya que no puede utilizar directamente los datos internos.
- Que en caso de que la clase cambie su estructura interna, pero no su interfaz pública, todo el código de los usuarios basado solo en la interfaz pública no tendrá que ser actualizado.

En C++ se cuenta con las palabras reservadas: **private** (privado), **protected** (protegido), y **public** (público) para controlar el acceso a los miembros de una clase, métodos y atributos. Si se omiten las palabras reservadas, por defecto en C++ los miembros se consideran **private** (privados). Cuando se utiliza alguna de las palabras reservadas seguidas de **:** todos los miembros declarados después tendrán las restricciones especificadas por el modificador de acceso. El siguiente código muestra un ejemplo del control de acceso de una clase. En la figura 2 se muestra un ejemplo gráfico de la clase Time.

Code 1: Time class

```
1  // Specification file (time.h)
2  class  Time {
3          public:
4                  void Set ( int hours, int minutes, int seconds);
5                  void Increment ();
6                  void Write ()  const;
7                  // Constructor
8                  Time ( int initHrs, int initMins, int initSecs);
9                  Time ();             // Default constructor
10
11         private:
12                 int hrs;
13                 int mins;
14                 int secs;
15
16         protected:
17                 //Miembros protegidos
18 };
```

Los modificadores de acceso tienen las siguiente restricciones:

public: Los miembros declarados **public** (públicos) estarán accesibles para cualquiera dentro del programa.

privade: Un miembro declarado **private** (privado) puede ser accedido por un objeto de esa clase sólo desde los métodos de dicha clase. Esto significa que no puede ser accedido por los métodos de cualquier otra clase, incluidas las subclases, ni por funciones externas de la aplicación incluida la función *main*.

protected: Un miembro declarado **protected** (protegido) se comporta exactamente igual que uno privado para las funciones externas o para los métodos de cualquier otra clase, pero actúa como un miembro público para los métodos de sus subclases.

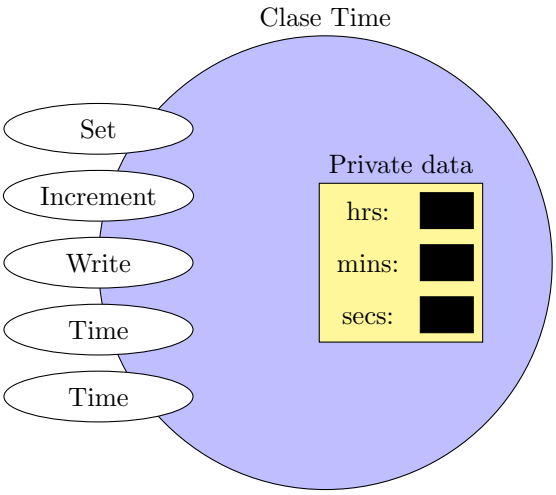

Figure 2: Control de acceso a la clase Time

1.3 Herencia

Una de las principales características de la Programación Orientada a Objetos es la *herencia*; La cuál provee un mecanismo para definir una nueva clase que añada nuevas características a una clase existente. La *herencia* es el mecanismo por el cual una clase adquiere (hereda) las propiedades de otras clases (métodos y atributos).

Con la herencia todas las clases están clasificadas en una jerarquía. La clase de la cual se heredan las propiedades se dice que es la clase base o superclase. La clase que hereda las características de alguna clase se le llama clase derivada o subclase. Cada clase tiene su superclase, y cada clase puede tener una o más subclases. Las clases que están en la parte inferior de la jerarquía heredan las propiedades de las que se encuentran en niveles superiores. También se dice que la clase derivada está especializada mediante la adición de propiedades que le son propias. En la figura 3 se muestra una jerarquía de clases. La clase Vehicle es la superclase de las clases Wheeled vehicle y Boat. Las clases más especializadas de la jerarquía son las clases Two-door y Four-door que heredan todas las características de la clase: Car. La clase Car, a su vez, hereda todas las propiedades de la clase Wheeled vehicle.

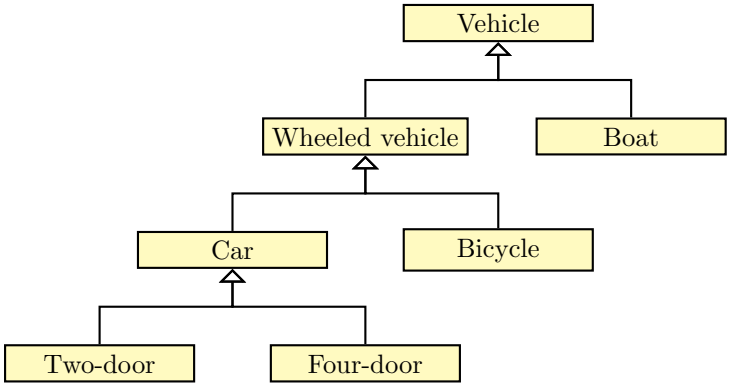

Figure 3: Jerarquía de clases

Una jerarquía de clases muestra cómo las clases se derivan de otras clases más simples heredando su comportamiento. En los lenguajes de programación orientados a objetos es común ver jerarquías de clases para describir la herencia.

La sintaxis para definir una nueva clase derivada es la siguiente:

```
class clase_der: [{private|protected|public}] clase_base_1
                 [,[{private | protected | public}] clase_base_2] ...
{
        //cuerpo de la clase derivada
};
```

El acceso a los miembros de una clase base son controlados por los modificadores de acceso **private, protected o public**. La clase base puede considerarse privada (private), protegida (protected) o pública (public). Por omisión se asume que es privada.

Por ejemplo:

```
class A {};                 //clase base
class B : private A {};     //clase base privada
class C : protected A {}; //clase base protegida
class D : public A {};      //clase base pública
```

Los modificadores de acceso en la clase base tienen las siguientes características:

private: Cuando la clase base es declara privada (**private**) todos sus miembros públicos (**public**) y protegidos (**protected**) pasan a ser privados (**private**) en la clase derivada.

protected: Si en una derivación la clase base es declarada protegida (**protected**) todos sus miembros públicos (**public**) y protegidos (**protected**) pasan a ser protegidos (**protected**).

public: Este modificador de acceso no tiene ningún cambio en los atributos de la clase, esto es, los miembros declarados públicos (**public**) siguen siendo públicos, los declarados protegidos (**protected**) siguen siendo protegidos, y los declarados privados (**private**) siguen siendo privados (**public**) en la clase derivada.

A continuación presentaremos un ejemplo de herencia, en el cual definiremos una clase derivada a partir de la clase base Time descrita en el código 1

```
// Specification file ('exttime.h')
#include   'time.h'
enum ZoneType{EST, CST, MST, PST, EDT, CDT, MDT, PDT};

class ExtTime : public Time // Time is the base class
{
public:
    void Set(int hours, int minutes, int seconds, ZoneType timeZone);
    void Write ()  const;
    ExtTime(int initHrs, int initMins, int initSecs, ZoneType initZone);
    ExtTime();
private:
          ZoneType  zone; // Additional data member
};
```

En la figura 4 se muestra una representación gráfica de la clase ExtTime. Se puede observar que los métodos Set y Write fueron sobreescritos, esto es redefinidos en la clase base. Cuando se mande a ejecutar el método Set o Write sobre un objeto ExtTime se ejecutará el redefinido en esta clase. El método Increment de la clase Time pasa a ser un método público de la clase ExtTime.

Los constructores de las clases bases no son heredados por sus clases derivadas, por lo cual definimos los constructores de la clase ExtTime. Es posible llamar al constructor de la clase base desde el constructor de la clase derivada para no reescribir la inicialización de las variables doble vez. Esto se hace poniendo **:** después de la definición de los parámetros del constructor de la clase, y posteriormente llamar al constructor de la clase base con los respectivos parámetros. Cómo ejemplo mostraremos una posible definición del constructor de la clase ExtTime.

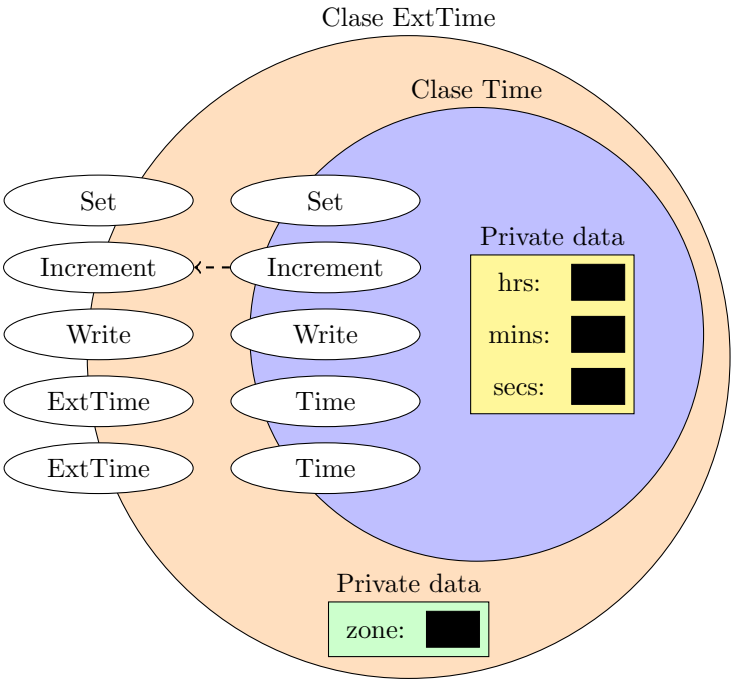

Figure 4: Definición de la clase derivada ExtTime a partir de la clase base Time, herencia.

```
1  ExtTime::ExtTime(/* in */ int initHrs,  /* in */ int initMins,
2                   /* in */ int initSecs, /* in */ ZoneType initZone )
3                   : Time (initHrs, initMins, initSecs)
4  // Constructor initializer
5  // Pre: 0 <= initHrs <= 23 && 0 <= initMins <= 59
6  //           0 <= initSecs <= 59 && initZone is assigned
7  // Post: zone == initZone && Time set by base class constructor
8  {
9      zone  = initZone;
10 }
```

1.4 Polimorfismo

El polimorfismo (facultad de asumir muchas formas) se refiere al comportamiento adecuado de los métodos de un objeto en una jerarquía de clases, independientemente del tipo de medio (puntero o referencia a una clase) empleado.

Cuando se tiene una jerarquía de clases es posible declarar un puntero de una clase base e instanciar un objeto de una clase derivada. Por ejemplo, en la figura 5 se se muestra un ejemplo de una jerarquía de clases.

Las clases A y B heredan de la clase base Base. El código 2 muestra una posible implementación en C++ para la jerarquía de clases de la figura 5. En la función main se declaró un arreglo de tres punteros del tipo de la clase Base. Pero al momento de instanciar los objetos, uno se instancia con la clase Base, otro con la clase A, y el último con la clase B. Posteriormente se llama al método msg() con cada uno de estos punteros. En este ejemplo el comportamiento que se obtendrá al llamar cada uno de estos métodos es la ejecución del método msg() de la clase Base.

Code 2: Jerarquía de clases de la figura 5.

```
1  class Base{
```

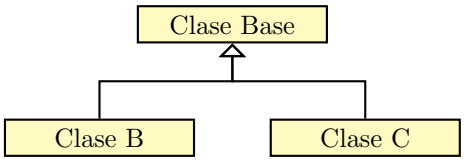

Figure 5: Jerarquía de clases

```
2         public: void msg(){  cout << "Clase␣Base" << endl;      }
3 };
4
5 class A: public Base{
6         public:     void msg(){ cout << "Clase␣A" << endl;      }
7 };
8
9 class B: public Base{
10        public: void msg(){     cout << "Clase␣B" << endl;      }
11 };
12
13 int main(){
14        Base *base[3];
15        base[0] = new Base();
16        base[1] = new A();
17        base[2] = new B();
18
19        for(int i=0; i < 3 ; i++)  base[i]->msg();
20 }
```

Si lo que se desea es llamar al método msg() de la clase a la cual pertenece el puntero implementado es necesario utilizar la palabra reservada **volatile** en la declaración de los métodos. De esta forma es posible tener un comportamiento de polimorfismo. Este comportamiento se muestra en el código 3.

Code 3: Jerarquia de clases con polimorfismo.

```
1  class Base{
2         public: virtual void msg(){
3                 cout << "Clase␣Base" << endl;
4         }
5  };
6
7  class A: public Base{
8         public: virtual void msg(){
9                 cout << "Clase␣A" << endl;
10        }
11 };
12
13 class B: public Base{
14        public: virtual void msg(){
15                cout << "Clase␣B" << endl;
16        }
17 };
18
19
20 int main(){
21        Base *base[3];
```

```
22          base[0] = new Base();
23          base[1] = new A();
24          base[2] = new B();
25
26          for(int i=0; i < 3 ; i++)   base[i]->msg();
27  }
```

En el ejemplo anterior la llamada al método msg() corresponde con la clase con la cual fue implementado el puntero (instanciado el objeto), y no con el cual fue declarado. Una posible aplicación de esta técnica puede ser una método N que reciba un puntero a una clase C y llamar a su método M. Si se tiene una jerarquía de clases con la clase C como clase base, y algunas clases que heredan de ella, digamos las clases A y B, se pueden crear punteros de la clase C que instancien objetos tipo A y B. Supongamos que el método M ha sido declarado virtual. Al llamar al método N, que recibe como parámetro un puntero de la clase C, se le pueden pasar punteros que instancian objetos A y B y llamar al método M. En este caso la ejecución del método M dependerá del tipo con el cual fue instanciado el puntero y se ejecutará ya sea el método N de la clase A, B o C, teniendo un comportamiento polimorfico.

References

[1] F. J. C. Sierra, *Programación orientada a objetos con C++*. Ra-Ma, 5 2003.

[2] H. . P. D. . Deitel, *C++ How to Program (5th Edition)*. Prentice Hall, 5 ed., 1 2005.